An Introduction to YADAS

Todd L. Graves[*]
Los Alamos National Laboratory

October 23, 2003

1 What is YADAS? And where can I get it?

YADAS is a software system for statistical analysis using Markov chain Monte Carlo (MCMC). It is written in Java and its source code is being distributed here (it continues to be developed). It was intended to be used by statistical researchers, and as such, it has always been a goal to make YADAS extensible enough to handle new models of forms not yet envisioned. Open source distribution is a vital part of the extensibility goal. However, it is one thing to have a system with no limits to its extensibility, and another thing altogether to have useful components available that make extensions easy. The `BasicMCMCBond` construct, with its argument functions and a small set of likelihood functions, enables specification of most models without requiring the user to write a great deal of new code. The `MCMCUpdate` interface allows users to update the parameters in their MCMC algorithms in arbitrary ways and in particular `MultipleParameterUpdate`s make it easy to propose Metropolis–Hastings moves in arbitrary directions. Metropolis–Hastings moves of multiple parameter simultaneously are powerful and intuitive ways to improve convergence of difficult MCMC algorithms.

The emphasis of YADAS is on Metropolis and Metropolis-Hastings moves. This aspect of YADAS runs contrary to the instincts of most Bayesians, whose first impulse is to implement Gibbs sampling using exact conditional distributions. The choice to concentrate on the Metropolis algorithm is more than a tribute to its Los Alamos origins, however. Avoiding Gibbs sampling frees analysts from the responsibility of evaluating and coding full conditional distributions (instead, the analyst has only to specify the terms in the unnormalized posterior distribution). Furthermore, the same algorithms that are used to compute acceptance probabilities for simple Metropolis moves can also be used to compute acceptance probabilities for more complex moves should they prove necessary. There is no incentive for analysts to force-fit their prior knowledge into conjugate forms. The only downside we can think of is that Metropolis steps require users to tune step size parameters, but this process is usually quite straightforward.

Many people are concerned with computation speed issues when they hear that YADAS is written in Java. Java's speed is underrated, and in any case, human time required to write an application is invariably much more precious than computational time. Still, the general-purpose algorithms used in YADAS will slow things down relative to special purpose code. If your problem is large enough that optimization with respect to speed is critical, YADAS will not be a feasible solution, at least in the 2003 era of Moore's Law. But you knew that already.

Several other general–purpose MCMC tools are available. First, WinBUGS (Spiegelhalter, D. J., A. Thomas, N. G. Best, and W. R. Gilks (1996), *BUGS: Bayesian inference Using Gibbs Sampling, Version 0.5, (version ii)*; see http://www.mrc-bsu.cam.ac.uk/bugs/welcome.shtml) deserves special notice. Our feeling

[*]Thanks to Vivian Romero and Yvonne Armijo, for their invaluable help.

is that if a problem can be handled by WinBUGS, then one should normally use WinBUGS to solve it. However, we believe that researchers who develop new models will have an easier time extending YADAS's open source code than WinBUGS. Bassist (Toivonen, H., H. Mannila, J. Seppanen, and K. Vasko (1999), *Bassist User's Guide For Version 0.8.3*; see http://www.cs.helsinki.fi/research/fdk/bassist/) is another good package; unfortunately, it appears that it is no longer being supported. It converts code written using a model description language into C++ code. It has good extensibility since it can use C++ functions to define model parameters, but it is limited in its capability for generating different sorts of MCMC algorithms. HYDRA (Warnes, G. R. (2002), *HYDRA: a Java library for Markov Chain Monte Carlo*, Journal of Statistical Software, Volume 07 Issue 04, 03/10/02) is also written in Java and distributed open-source. It does not seem to have any limits in its extensibility, but neither does it provide classes that make it easy to specify new models (one has to define the posterior distribution from scratch) or to define new MCMC algorithms, although several special algorithms are included. Flexible Bayesian Modeling (FBM) is a set of UNIX based tools for many sorts of Bayesian regression models. (Neal, R. (2001), Software for Flexible Bayesian Modeling and Markov Chain Sampling; see http://www.cs.toronto.edu/ radford/fbm.software.html).

Finally, YADAS is an acronym for "yet another data analysis system", and is pronounced as if it were a contraction of "yada yada yada". We do not anticipate any compliments on our sparkling wit.

In this documentation, we will begin by reviewing Markov chain Monte Carlo ideas in §1.1, and by reviewing object-oriented programming in Java in §1.2. We introduce the key components in any YADAS analysis in §2, and discuss how we normally deal with poor mixing in MCMC in §3. We present a number of examples in §4, each of which illustrates an advanced topic in YADAS.

1.1 Review of MCMC ideas

It is not clear to me that YADAS is going to be useful to someone unfamiliar with the basic ideas of Markov chain Monte Carlo (MCMC), but if you fall into that category, a good reference is A Gelman, JB Carlin, HS Stern and DB Rubin, *Bayesian Data Analysis*, CRC Press, 1995. Still, we will motivate design decisions made in YADAS here by describing MCMC in a particular way.

MCMC is a method of numerical integration: if the information (derived from prior knowledge and from experimental data) about an unknown parameter θ is contained in a posterior distribution $f(\theta)$ with respect to some measure μ, the integral $I = \int g(\theta)f(\theta)d\mu(\theta)$ provides a point estimate of the quantity $g(\theta)$. If this integral cannot be estimated in closed form, one obtains a sequence $\theta^1, \ldots, \theta^B$ of dependent samples from the distribution f, and estimates I by $\hat{I} = B^{-1}\sum_{b=1}^{B} g(\theta^b)$. To obtain the dependent sequence, one uses MCMC, in which one generates a Markov chain whose stationary distribution is f and which mixes as efficiently as possible (in the sense that successive samples are as uncorrelated as possible). To define an MCMC algorithm, one needs a method of obtaining the next sample θ^{b+1} from the current sample θ^b. A useful idea is the Gibbs sampler, which divides θ into several components $\theta_1, \ldots, \theta_k$, begins by sampling a new value of θ_1 from its conditional distribution given the values of the other components.

The Metropolis–Hastings algorithm, on the other hand, consists of a proposal distribution $T(\theta, \theta')$ according to which a new value θ' is proposed given the new current value θ. This proposal can either be accepted, in which case the chain moves to θ', or it can stay at θ for another iteration. To ensure that the resulting Markov chain has the correct stationary distribution f, the probability that the move to θ should be accepted can be taken to be

$$\frac{f(\theta')}{f(\theta)} \frac{T(\theta', \theta)}{T(\theta, \theta')}.$$

Several things can be seen from this acceptance probability formula. First, one only needs to be able to evaluate $f(\theta)$ up to a multiplicative constant, because ratios of f's evaluated at two different values are what is of interest. Second, a useful special case (the Metropolis algorithm) is the case where the proposal distribution is reversible ($T(\theta, \theta') = T(\theta', \theta)$). Finally, one can write very general software by developing an alphabet for evaluating f's, and another alphabet for constructing proposal densities T and evaluating ratios of those proposal densities. This is the approach taken in YADAS. The most common sort of proposal is, as in Gibbs sampling, to divide θ into components (where each component is one-dimensional real valued). θ' is then constructed from θ by generating a random standard Gaussian variable Z, and adding $s_k Z$ to the kth component of θ. Here $s_k > 0$ is a tunable step size parameter, and the algorithm consists of one move of this sort for each component. This sort of proposal is reversible, so it can be called a Metropolis step. As we will detail later, different problems call for more specialized proposal distributions, in particular when f induces parameters to be highly correlated with one another, or when certain of the parameters have discrete or mixed continuous-discrete distributions.

1.2 Some notes on object-oriented programming in Java

On the other hand, probably most people interested in YADAS have little to no experience in object-oriented programming, in Java or otherwise. Some good references are C. S. Horstmann and G. Cornell, *Core Java 2: Volume 1- Fundamentals* and *Core Java 2: Volume 2- Advanced Features*, and B. Eckel, *Thinking in Java.*

Object-oriented programming consists of creating and manipulating *objects*. Objects are data structures that can also contain functions (called *methods*) as part of their definition. The description of what an object and all objects like it have in common is called a *class*, and examples of objects in that class are called *instances* of the class. Classes can be organized in *class hierarchies*, where a *subclass* of another class (a *superclass*) is similar to the superclass but is typically more specific: it can have additional data or methods in its definition, and it can define its methods in ways different than its superclass. (Unlike in some other object-oriented languages, classes in Java can have only one superclass). Another important concept is the *interface*: an interface is a collection of methods, and a class is said to *implement* that interface if it contains definitions of all these methods. One way that object-oriented programming is powerful is that one can place into an array several objects that belong to a general superclass or that implement the same interface, and then one can operate on those objects using methods common to all of them. Each object will behave in its own way in response to those methods. Java also contains primitive data types such as integers, reals, and characters, and these are not objects.

In Java, objects (and primitives) must be first *declared*, which sets aside storage space for them before actually defining their initial values, and then they can be *initialized*, in which they are given initial values. Suppose `Myclass` is the name of a class an one wishes to define an instance of this class named `Myobject`. Declaration of `Myclass` appears as `Myclass Myobject;` in Java code. After this, one can include a statement `Myobject = new Myclass();` to initialize the object. What has actually been done here is a call to the *constructor* method of the `Myclass` class; this method constructs a new instance of the class. Most often, some arguments will appear insides the parentheses in the call to the constructor method. `new` is a very important keyword in Java.

Now suppose there is a method called `run()` defined in the `Myclass` definition. This method can be called for the `Myobject` object using `Myobject.run()`. Depending on the method definition, arguments may also appear inside the parentheses.

Java is *strongly typed*, so one cannot liberally mix integers with real numbers as in S-Plus. A potentially annoying consequence of this in YADAS is that all arguments to likelihood functions must be real valued, even binomial data and sample sizes. Arrays start at zero, as in C, rather than at one. All statements must end with semicolons.

Java is a compiled language; the compiler is called `javac` and it must be run to convert `.java` files into `.class` files, which can then be run using the `java` command.

1.3 Installation instructions

First, you need to have Java (version 1.2 or later) available on your system. If you don't, download it from `java.sun.com`; the Standard Developer's Kit (SDK) is sufficient.

Then, download the YADAS source code in `yadas.tar.gz` or the `.jar` file. Uncompress and untar the source code file into the directory you want to keep your YADAS code. The files should be in a directory called `direc/gov/lanl/yadas/`; the value of `direc` is your choice. Another way of saying this is that YADAS code is part of a package called `gov.lanl.yadas`. The standard for naming Java packages is to reverse one's web URL.

On Unix systems, you will often need to edit your `.cshrc` file or analogous file to change your `CLASSPATH`. (I understand that Linux systems have a similar file with a different name.) It should contain '.' and the directory `direc` in which you place `gov/lanl/yadas`, or the `.jar` file. For example, my own `.cshrc` file contains a line

```
setenv CLASSPATH .:/home/tgraves/Java:/home/tgraves/Java/MCMC:/home/tgraves/Java/colt.jar
```

indicating that any `.class` files in `/home/tgraves/Java` or `/home/tgraves/Java/MCMC` will be available from wherever I run a Java application; I have also downloaded the COLT software package (which I recommend to you as well: http://hoschek.home.cern.ch/hoschek/colt/index.htm) and use their .jar file in `colt.jar`. I actually keep `gov/lanl/yadas` inside `/home/tgraves/Java`. The syntax for defining a classpath environment may vary depending on your version of Unix.

The Windows analogue is the `AUTOEXEC.BAT` file, which also needs to have a classpath definition. For example, if Java was installed to `d:`
`j2sdk1.4.0` and you placed the `yadas.jar` file in the directory `d:`
`Java`, the following line should appear in your `AUTOEXEC.BAT` file:

```
SET CLASSPATH=.;d:\\j2sdk1.4.0\\lib;d:\\Java\\yadas.jar
```

The '.' at the beginning is necessary if you want to run any classes in your current directory. Alternatively, if you downloaded the YADAS source code and placed the `gov` subdirectory inside `d:`
`Java`, a potentially appropriate line in `AUTOEXEC.BAT` is

```
SET CLASSPATH=.;d:\\j2sdk1.4.0\\lib;d:\\Java
```

It is also allowable to have both the `Java` directory and the `yadas.jar` jar file in your classpath. I don't remember exactly why I needed to list the Java SDK library in my Windows version and not in my Unix version. I have no experience using Java on a Macintosh, OS X or otherwise, but I hope to remedy this omission before long.

When writing your own YADAS applications, you need to put the line `import gov.lanl.yadas.*` near the top of the file.

To run any of the examples obtained from the YADAS web site after you have unzipped and untarred the `Examples.tar.gz`, move to the appropriate numbered subdirectory of `Examples`. If the directory does not already contain the `.class` file for your desired application (e.g. `OneWayAnova.class`), type the command `javac OneWayAnova.java` at the command line (either DOS or Unix). If the `.class` file is there, you can run the application with the command `java OneWayAnova 10000`, if `OneWayAnova` is the desired application and if you wish to run 10000 iterations. Among the things that can go wrong: `javac` or `java` may not be in your path, and the current directory (`.`) may not be in your classpath.

1.4 Alternative interfaces that may appear in the future

It is currently standard to run YADAS applications by writing Java code. Recently, we ran our first application from Jython (see www.jython.org), a Java implementation of the scripting language Python (see www.python.org). Beyond Python's power, Jython also makes it quite easy to utilize Java code. At the moment, the Jython approach contains few usability advantages over the Java approach, but it is an area of further work. We would also like to develop interfaces from R, probably through the omegahat project (see www.R-project.org or www.omegahat.org) and MATLAB. If you are interested in contributing to these efforts, please contact yadas@lanl.gov.

2 The basics

On the following pages we will discuss the fundamental concepts in YADAS: the objects and methods that appear in essentially all applications. The first example we will discuss is a one-way ANOVA example, where normally distributed data Y_{ij} have means μ_i and known standard deviation σ ($1 \leq i \leq I, 1 \leq j \leq J$); the μ_i are normally distributed with mean θ and known standard deviation δ, and θ has a flat prior on $(-\infty, \infty)$. This problem is straightforward to analyze with Gibbs sampling or alternatively one can even find the posterior distribution analytically. However, it also illustrates many of the fundamental concepts of YADAS. It would be helpful to view the source code for `OneWayAnova.java`, and possibly also the data files, while reading these sections. We begin by introducing the classes we use to import data from input files, `DataFrame` (§2.1) and `ScalarFrame` (§2.2). Understanding these classes is necessary for reading my YADAS code. We then introduce the `MCMCParameter` (§2.3) class: every unknown quantity that gets updated in the course of an MCMC algorithm is an `MCMCParameter`. Next we introduce the `BasicMCMCBond` class, which is responsible for most of the power and flexibility of the model specification capabilities of YADAS. At that point we will understand the one-way ANOVA example and be able to work many other simple examples.

2.1 DataFrame

Most YADAS applications are written with no hard-wired constants: instead, all inputs are stored in files. `DataFrame` is the most common class used to import data from a file. An example of a input file that can be read into a `DataFrame` is as follows:

```
6
y|group
r|i
0.5|0
0.8|0
0.3|1
0.4|1
1.2|2
0.9|2
```

The first line is the number of lines of data in the file. The second line is a list of variable names, separated by pipes (|). The third line contains information about the type of the variable ('r' for real-valued, and 'i' for integer; 's' for string is also possible, but as of this writing is rarely used). The variable types are also separated by pipes. The fourth through last lines contain the data: in this example, the first column of data contains the values of the real-valued 'y' variable, while the second column of data contains the values of the integer-valued 'group' variable. Suppose this content is in the file `data.dat`. These data can be read into a YADAS application using the code

```
DataFrame d = new DataFrame (``data.dat'');
```

This code defines a new data frame called `d`. Once this is done, one can access the `y` variable using `d.r(``y'')`, or the `group` variable using `d.i(``group'')`. This is similar to `d$y` or `d$group` in (S-Plus).

All the variables in a `DataFrame` have the same length, so an application will often require multiple `DataFrame`s. For example, in `OneWayAnova.java`, a `DataFrame d` contains one row of data for each data point, and another `DataFrame d2` contains one row of data for each level of the grouping variable. The scalar variables could also be stored in a `DataFrame`, with a single row of data, but scalar data files tend to be easier to read when stored in a `ScalarFrame` (see §2.2) instead.

Let `d` be a `DataFrame`.

- `d.r(``realname'')` returns an array of real numbers whose values are stored under the name "realname" in `d`.
- `d.i(``intname'')` returns an array of integers whose values are stored under the name "intname" in `d`.
- `d.length()` returns the integer length of the variables in `d`.
- `d.r(1.5)` returns an array of real numbers, all of whose values are 1.5. The length of this array is the same as the length of all the variables in `d`.
- `d.i(9)` is analogous, but it returns an array of integers.
- `d.u()` returns an array of integers of the same length as `d` whose first element is zero, the second element is one, and so on up to the last element whose value is the length minus one.

2.2 ScalarFrame

The file `Ex1scalars.dat` used in the one-way ANOVA example is an example of a `ScalarFrame`, a tool used in YADAS to read input files consisting of scalars. It is also possible to store scalars in `DataFrame`s, but in large problems with many scalars, these files can get hard to read. Each line in a file to be read by `ScalarFrame` contains a variable name and a value for that variable, separated by a pipe. For example, the file `Ex1scalars.dat` begins with

```
theta|246.5
thetamss|3
sigma|2
sigmamss|1
delta|2
```

All variables in `ScalarFrame`s are assumed at first to be real-valued. If `d0` is a `ScalarFrame` in a YADAS application, one can extract the variable named `theta` by calling `d0.r(``theta'')`. This method returns an array of real numbers of length one, not a real scalar (Java makes a distinction). If one wants a `ScalarFrame` to store an integer parameter named `n`, one can define the parameter in the same way as one would a real parameter, and then call a command such as `d0.i(``n'')`. Again, this is an integer array of length one.

2.3 MCMCParameter

After reading the contents of input files into `DataFrame`s and `ScalarFrame`s, the next step is to define `MCMCParameter`s. Parameters are the quantities that are updated in the course of the MCMC algorithm. Any quantity whose posterior distribution is a target of the MCMC algorithm must be stored in an `MCMCParameter` or one of its subclasses, and sometimes it is also appropriate to define constant parameters. To initialize a parameter, one needs an array of (real) initial values, an array of (real) step sizes that will most often be used in Metropolis steps, and a (string) file name. For example, the definition of the random effects μ in the one-way ANOVA example is

```
mu = new MCMCParameter (d2.r("mu"), d2.r("mumss"), direc + "mu"),
```

Here `d2` is the `DataFrame` containing the data in the file `Ex1mu.dat`. The variable called `mu` in this file stores the initial values for the `mu` parameter, and the variable called `mumss` stores the step sizes for this parameter. Note that the parameter contains three components (i.e. there are three main effects μ_0, μ_1, and μ_2) and three step sizes, one for each component. `direc` contains a directory name in which the input files are kept and to which the MCMC output will be sent; in particular, the samples of the `mu` parameter will be sent to a file called `mu.out`.

Later we will discuss the `MCMCUpdate` interface, according to which all the ways of updating parameters are defined. For the moment, `MCMCParameter`s are the simplest example of updates. Contained in the definition of the `MCMCParameter` class is the general componentwise Metropolis algorithm. In other words, when the `update()` method of a parameter is called, YADAS loops over the components of the parameter. For each component, it proposes a Gaussian move centered at the current value of the component and with that component's step size as the standard deviation of the proposal distribution. This move is then accepted with the appropriate Metropolis probability calculated from the ratio of the posterior distribution for the new and old values of the parameters, and YADAS moves on to the next component. To be precise, denote by θ_{-i} the set of current values of all unknown parameters in the model excepting θ_i. When the algorithm attempts to update θ_i, the proposed new value θ_i' is constructed by $\theta_i' = \theta_i + s_i Z$, where s_i is the step size for θ_i and where $Z \sim N(0, 1)$. If p denotes the (possibly unnormalized) posterior density function, written with two arguments (θ_i and θ_{-i}), this move is accepted with probability $p(\theta_i', \theta_{-i})/p(\theta_i, \theta_{-i})$. Otherwise θ_i remains unchanged for this iteration of the algorithm, and in either case we move on to trying to update the next component of θ.

`MCMCParameter`s are not the most powerful or challenging pieces of YADAS. Don't let the need to specify step sizes scare you away; YADAS provides output regarding acceptance rates for the Metropolis steps, and it is usually straightforward to tune the step sizes to attain acceptance rates of roughly 40%, which will often correspond to good mixing of the chain.

YADAS also contains the capability of updating parameters on the log scale: a `MultiplicativeMCMCParameter` θ_i obtains its proposal through the mechanism $\theta_i' = \exp(s_i Z)\theta_i$, where Z is standard Gaussian. In some cases it may seem natural that a parameter's variation is multiplicative rather than additive, although it is not entirely clear that this is ever necessary. Try changing the definition of `delta` in the one-way ANOVA example to make it a `MultiplicativeMCMCParameter` (i.e. change the line `delta = new MCMCParameter...` to `delta = new MultiplicativeMCMCParameter...`), recompile, play with the step size, and explore whether mixing is improved. Clearly only positive parameters (or, rarely, negative parameters) should be defined as `MultiplicativeMCMCParameter`s.

Similarly, `LogitMCMCParameter` updates a probability parameter by generating the proposal additively on the logit scale. We have found `LogitMCMCParameter`s useful in reliability applications in which a probability parameter is close to one or zero. For example, see §4.4 for an analysis of system reliability.

2.4 BasicMCMCBond: how to express a model

One of the key characteristics of the software architecture of YADAS is the `BasicMCMCBond` structure. We think that this construct makes it as easy as possible to define most statistical models. It also makes it easy to make small changes to existing analyses, for example by adding another level to a hierarchy or adding a prior distribution to a quantity that had previously been fixed. It is also very easy to change distributional forms and link functions.

In YADAS, an `MCMCBond` is a term in the unnormalized posterior distribution (`MCMCBond` is actually an interface). One computes the unnormalized posterior density function by multiplying all the bonds together: for example, think of the prior density function as one bond, and the likelihood function as the second bond. More generally, many parameters can have independent prior distributions, each of which can be captured in a bond, and multiple sources of data with different likelihoods can be encoded in other bonds. The purpose of a bond, then, is to compute a desired function of unknown parameters.

Nearly all problems can be handled using only one type of bond, the `BasicMCMCBond`. A `BasicMCMCBond` consists of three parts:

1. an array of parameters,
2. an array of `ArgumentMaker`s, which are objects that compute functions of these parameters, and
3. a `Likelihood` function, which takes the output of the argument functions and returns the value of the log-likelihood for that term in the posterior.

YADAS is set up to use as few `Likelihood` functions as possible (e.g. Gaussian, Gamma, Binomial, and so forth) and to put most of the variation between problems into `ArgumentMaker`s.

At this stage you should refer to the file `OneWayAnova.java`. In this problem, the posterior contains four bonds: the data bond (or likelihood), and the priors for the random effects, the data standard deviation, and the random effect standard deviation.

2.4.1 Likelihoods

For two reasons, the name `Likelihood` is unfortunate: first, it is used in prior terms as well as likelihood terms; and second, it actually computes the log-likelihood function rather than the likelihood. However, the construct is very useful regardless of its poorly chosen name. A small number of `Likelihoods` enable the bulk of analyses. For example, consider the Gaussian likelihood (see the code in `Gaussian.java`). A likelihood's vital characteristic is its `lik` method, which takes a two-dimensional array of reals as input and returns a single real. In Gaussian, the first "column" in the input array is the vector of "data" (call them y_i), the second column is the vector of means (call them μ_i), and the third column is the vector of standard deviations (call them σ_i). The likelihood computes the function

$$\sum_i \{-\log(\sigma_i) - (y_i - \mu_i)^2/2\sigma_i^2\}.$$

In Gaussian, the function requires the same number of data points, means, and standard deviations, and most Likelihoods behave similarly, even though in many problems, all the means or standard deviations are identical.

"Constants" are typically computed in Likelihood functions. This leads to some loss in performance, but that is the price of generality. In the Gaussian example, if one is considering a change to the mean parameter, the term involving $-\log(\sigma_i)$ is the same for the current parameter set and for the proposed parameter set, and will just be canceled out, but we compute it anyway. A more extreme example is if the standard deviation is known, in which case most Bayesians will think of this term as a constant and will recoil at the idea of computing it and subtracting it from itself. In today's analysis, a term may be a function of constants, but in tomorrow's analysis, some of those constants may be unknown parameters. Computing these constant terms allows us to get by with a single Gaussian likelihood function. (Obviously, you should feel free to write your own `GaussianFixedSD` likelihood function that refrains from computing the constant term if it makes you feel better.)

Here are the likelihood functions that you are likely to need. All of these with the exception of `MultivariateNormal` take rectangular arrays. We describe the meanings of the columns of the array. The output of the `lik` method is the sum, over rows, of the function applied to the elements in each row.

- `Gaussian`. Three arguments: data y, mean μ, and standard deviation σ. Computes $-\log\sigma - (y-\mu)^2/2\sigma^2$. Again, we stress that the Gaussian distribution is parameterized in terms of its *standard deviation*.
- `Gamma`. The three arguments are data y, shape parameter α, and scale parameter θ, so that the mean of y is $\alpha\theta$.
- `Poisson`. The two arguments are the data y and the mean λ. A source of confusion for `Poisson` and `Binomial` is that all likelihood functions expect all their arguments to be real, not integer, valued. This holds even for Poisson data, and Binomial data and sample sizes. On the other hand, it is legal to use noninteger values for these should you feel the urge.
- `Binomial`. The three arguments are, in order, the number of successes x, the sample size n, and the probability p. See the note for the Poisson distribution for a warning.
- `Beta`. The three arguments are the data y and the two parameters a and b. The mean of y is $a/(a+b)$.
- `StudentT`. This distribution takes four arguments: the data y, the mean μ, the scale parameter σ, and the number of degrees of freedom ν. For large ν, the standard deviation is approximately σ. Note that this includes the Cauchy distribution, when $\nu = 1$.
- `Uniform`. The first argument is the data y, the second argument is the lower limit a, and the second argument is the upper limit b. The uniform likelihood can be used to state that a parameter is bounded between two values, for example in interval censoring problems.
- `MultivariateNormal`. We have experimented with some multivariate normal applications, but we have not bundled the code with this distribution because it uses third party software for matrix manipulation. The `MultivariateNormal` functionality should appear in a version of YADAS soon.
- `NegativeBinomial`. The negative binomial distribution can be used to model count data that are overdispersed relative to the Poisson distribution; see McCullagh and Nelder (1985, §6.2.3 in the second edition) and Graves and Picard (2002). The first argument is the data y, the second argument is the mean μ, and the third argument is an index parameter ϕ. The variance of y is equal to $\mu(1+\phi)/\phi$.
- `Weibull`. The Weibull takes three arguments: data y, scale parameter σ, and index parameter ϕ. We also supply a separate class `Weibull3` that allows an additional argument (the left endpoint of the distribution).
- `Dirichlet`. The Dirichlet distribution takes two arguments: the vectors of "data" (the probabilities) and the exponents in the Dirichlet prior.
- `InverseGamma`. Because conjugacy is no advantage in YADAS, we do not anticipate that the InverseGamma family will be used often. However, it is here: the arguments are the data, shape and scale parameters.

- `Hypergeometric`. For sampling from finite populations, the hypergeometric distribution is available. Its arguments are the number of successes in the sample, the size of the sample, the size of the population, and the number of successes in the population.

2.4.2 ArgumentMakers

The set of `Likelihood`s described earlier are powerful because of the use of `ArgumentMaker`s to transform parameters into arguments to the likelihoods. These are functions that act as if they take one or several parameters as inputs, and return an array to be fed into a `Likelihood` function. In principle, if our multidimensional parameter is θ, this allows us to use the Gaussian likelihood to specify that $f_0(\theta) \sim N(f_1(\theta), f_2(\theta)^2)$, for essentially any choices of f_1 and f_2 and for many choices of f_0. Naturally, these functions can access data as well as parameters. The three simplest and most commonly used argument functions are `ConstantArgument`, `IdentityArgument`, and `GroupArgument`.

- `ConstantArgument` ignores all the parameters that it is allowed to use to construct its output and instead returns the same constant vector each time it is needed. These are used for including data or fixed prior parameters in a bond. A `ConstantArgument` can be defined in several ways:
  - `new ConstantArgument (double[] x)` defines an argument function which returns the array of double precision numbers given in `x`. For example, in the one way ANOVA example, the data y are placed into a Gaussian likelihood function in this way.
  - `new ConstantArgument (double x, int n)` will return an array of length n, each of whose entries is x.
  - `new ConstantArgument (double x)` will return an array of length one, whose single entry is x.
- `IdentityArgument` is also very trivial. Suppose that the array of parameters in the bond is {`theta, mu, sigma`}. `new IdentityArgument (0)` reads the values of the 0th parameter (`theta`) and returns them unchanged. `new IdentityArgument (2)` does the same with the 2nd parameter, and this is `sigma` because arrays start at zero. Note the vital difference between `IdentityArgument`s and `ConstantArgument`s: `IdentityArgument`s read the values of parameters, and these can change in the course of the algorithm.
- `GroupArgument` is less trivial: its most common use is to take a parameter with a small number of components and lengthen it into a longer argument for a likelihood function. In the one way ANOVA example, each data point is assumed to have the same error standard deviation σ. The `sigma` parameter has only a single component, but when YADAS computes the Gaussian likelihood, it needs one of these standard deviations for each data point, and `GroupArgument` makes the appropriate array. A `GroupArgument` is defined by an integer serving the same role as in `IdentityArgument`, and by an array of integers I usually call an *expander*. This array plays the role of a subscripting vector in S-Plus. For example, suppose the parameter `mu` has three entries (μ_0, μ_1, μ_2). Suppose in a one-way anova problem that the first two data points belong to the first group, the next two data points belong to the second group, and the last two data points belong to the third group. Then, in S-Plus, one can define a vector `group <- c(1, 1, 2, 2, 3, 3)`, after which `mu[vec]` yields a vector $(\mu_0, \mu_0, \mu_1, \mu_1, \mu_2, \mu_2)$ that corresponds nicely with the data vector `y` in that `sum((y-mu[vec])^2)` is a residual sum of squares. Check out the examples of `GroupArguments` in the one-way ANOVA example. Again, the index of the first element in an array in Java is zero, so in YADAS, the group vector would be $(0, 0, 1, 1, 2, 2)$ instead of $(1, 1, 2, 2, 3, 3)$. A non-obvious use of `GroupArguments` is to place an autoregressive relationship on a vector of parameters: in particular, $\theta_i \sim N(\theta_{i-1}, \sigma^2)(i = 1, \ldots, n-1)$ can be defined using one `GroupArgument` with expander $(1, 2, \ldots, n-1)$ and another `GroupArgument` pointing at the same parameter but with expander $(0, 1, \ldots, n-2)$.

YADAS does not yet have the capability to apply an arbitrary function to generate the argument that plays the role of the data in a Likelihood. In general, if one wants to specify that $f(\theta) \sim G$, one needs to include the Jacobian of the transformation in the posterior, but this is not yet possible in general using YADAS, so at this stage you should not use an Argument function to generate the data argument unless its Jacobian is one.

2.5 Example 1: One-Way Anova

Here is the source code for a simple example. We have already discussed most of the example. See §1.3 for instructions for running the examples.

2.6 Example 2: FunctionalArgument

Our second example is also a one-way ANOVA example, but with the difference that the error standard deviation is now proportional to the mean: $Y_{ij} \sim N(\mu_i, \{\gamma\mu_i\}^2)$. Here γ is called a relative standard deviation, or RSD. See the source code in the file `OneWayAnovaRSD.java`, and/or click on the buttons below. Only a small change to the YADAS application code is necessary to analyze this model, but it does require us to introduce the `FunctionalArgument`, which is very powerful but not particularly natural to use. It allows us to specify that arguments to likelihood functions are arbitrary functions of the parameters. The user includes the function definition (in this case, $g(\mu, \gamma) = |\gamma\mu|$) in the definition of the argument. In the example, the bond contains two parameters, μ and γ. A `FunctionalArgument` is defined by five things:

- An integer that indicates how long the argument should be: for the example, there needs to be one standard deviation for each data point;
- An integer that indicates how many parameters are contained in the bond (in this case, two);
- An array of integers that indicate which of the parameters need to be "expanded" before being sent through the function. In this case, the zeroth parameter `mu` and the first parameter `gamma` are both the wrong length, so this array consists of both 0 and 1;
- A two-dimensional array of "expanders", much like those in `GroupArgument`. Here, we use the same group variable that we used in the mean argument to expand `mu` again, and we use the vector of all zeroes to expand `gamma`, just as in the previous example;
- a `Function`. In the example, the definition of the function is:

  ```
  new Function () { public double f(double[] args) {
   return Math.abs(args[0] * args[1]); }}
  ```

  The only part of this definition that needs to change for other applications is the part between the inner set of brackets (starting with `return` and ending with the semicolon). This code explains how to take the values in the expanded parameters and operate on them to get the output of the argument function. The function is called once for each data point. The ith time it is called, `args[0]` contains the value of the first parameter (`mu`) for the ith data point, and `args[1]` contains the value of the second parameter (`gamma`) for the ith data point. The function definition, then, says that the function multiplies `mu` by `gamma` and takes the absolute value for good measure, although both `mu` and `gamma` should be positive anyway.

`FunctionalArgument` has many uses; for example, it has been used to allow users to parameterize the gamma and beta distributions by their mean and variance. It can also be used to construct linear models, though if these linear models become too large, `LinearModelArgument`, described in the next section, is handier. A potential annoyance is that `FunctionalArgument` can force users to define parameters that are more naturally thought of as data (they do not get updated).

2.7 Example 3: LinearModelArgument

Somewhat like `FunctionalArgument`, `LinearModelArgument` is a YADAS construct which is difficult to use at first but which is powerful when one becomes accustomed to it. This class reads several columns from a `DataFrame`, interprets some of the columns as covariates and some as group labels for categorical covariates, adds up everything to get the linear predictor, and optionally runs it through a link function to get a transformed linear predictor. It is thus capable of calculating any quantity of the following form: the ith value is $h(\sum_{j=0}^{J} \beta_j X_{ij} + \sum_{k=0}^{K} \eta_{g_k(i)})$, where the βs and the ηs are allowed to be unknown parameters, whereas the Xs are data. Needless to say, this class can be used in generalized linear models as easily as ordinary linear models. Also, linear models are not restricted to modeling the mean of data; we can also have regression relationships in variances or other parameters. The most general constructor for `LinearModelArgument` takes the following arguments.

- A `DataFrame` that contains the covariates and/or group labels. The covariates must be labeled as real-valued, while the group labels must be integers. The `DataFrame` is allowed to contain real-valued columns that will not be used in generating the linear model (for example, the same `DataFrame` will usually contain the response variable in the regression). As of this writing, the frame can contain integer variables that do not appear in the linear model, but they must be listed in the frame after the variables that do.
- An integer that indicates whether or not the linear model contains an intercept (0 if no, 1 if yes).
- An integer indicating which of the parameters in the bond is the vector of regression coefficients. This integer behaves similarly to integers found in the `GroupArgument` and `IdentityArgument` discussions. Denote this parameter by $\beta = (\beta_0, \beta_1, \ldots)$. The first element in the vector is the intercept, if there is one. (We will still denote the first element by β_0, whether or not there is an intercept.) The remaining elements will be multiplied by real-valued covariates.
- The fourth argument is an array of integers whose purpose it is to map the real-valued variables in the `DataFrame` to the regression coefficients β so that they can be multiplied by each other. Suppose this array is $\{0, 1, 3\}$. If there is no intercept in the model, β_0 is to be multiplied by the zeroth real variable in the `DataFrame`, β_1 is to be multiplied by the first, and β_2 is to be multiplied by the third. If there is an intercept, β_1 goes with the zeroth real variable, β_2 with the first, and β_3 with the third. In other words, all the elements of the β vector must be used, while one can ignore real variable columns in the data frame.
- The fifth argument is another array of integers, and its purpose is to map categorical variables to parameters in the bond. Each categorical variable requires its own parameter. A categorical variable is represented in the data frame by an array of integer group labels; for example, suppose that the parameter is denoted by η and the group labels are $\{0, 0, 1, 1, 2, 2\}$; this means that the six values of the linear predictor contain, respectively, the terms $\eta_0, \eta_0, \eta_1, \eta_1, \eta_2$, and η_2. Suppose that this array of integers is $\{0, 4, 1\}$. This means that the zeroth integer-valued variable in the data frame is mapped to the zeroth parameter, the first variable is mapped to the fourth parameter, and the second variable is mapped to the first parameter. In other words, one cannot skip integer columns in the data frame, but one can ignore parameters. There is an asymmetry in the interpretations of the arrays that define the real covariates and the categorical covariates; I hope it will not be too confusing.
- Finally, one can optionally include a (inverse link) `Function` as in `FunctionalArgument` to transform the linear predictors.

Example 3, `LinearModelExample`, illustrates how to define a linear model. Click on the buttons to view the source code and data if you wish. In this example, we have two real covariates plus an intercept and two categorical predictors. The model is

$$Y_i \sim N(\beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \gamma_{g_1(i)} + \gamma_{g_2(i)}, \sigma^2),$$

where all three βs, all γs, and all δs have normal prior distributions, and σ has a Gamma prior distribution (no hierarchical models anywhere). For the data included with the examples, there are two γs and three δs, and the true values of the parameters are $\beta_0 = \beta_1 = 0, \beta_2 = 2, -\gamma_0 = \gamma_1 = 4, \delta_0 = -2, \delta_1 = 0, \delta_2 = 2$, and $\sigma = 0.5$. If you run this example with the data and prior parameters given with the examples, the MCMC will mix poorly as β_0 is highly negatively correlated with $\gamma_i + \delta_j$ for any i or j. (Otherwise, the results will be "right.") At this point in the tutorial, it is too soon to introduce measures that will improve mixing in this problem; `MultipleParameterUpdate`s with `OneUpOneDownPerturber`s should be useful, and we are at work on a general update step for linear models.

2.8 Writing your own Likelihoods

The supplied collection of `Likelihood`s should be sufficient for most analyses, but it is also not difficult to write your own. The most straightforward way to begin is with a supplied likelihood function, and modify it to compute another log-likelihood function. For example, to create a `Likelihood` called `MyLikelihood`, open (for example) `Gamma.java`, replace `Gamma` after `public class` by `MyLikelihood`, and eventually save the result in a file called `MyLikelihood.java` after changing the code inside the file.

A `Likelihood` object contains two methods, both called `compute`. The second method, which takes an array of integer indices as an argument in addition to the two-dimensional array of `double`s is deprecated, and you need not worry about changing it. The first `compute` method takes a two-dimensional array of numbers and processes them to output a log density value. Normally each row in the array is processed individually, and the results are added together (on the log scale); an exception is `AttritionLikelihood`, a model for randomly generated permutations which will be discussed later.

A technique the author has made use of is the judicious use of negative infinities. If a given piece of data has density zero at the associated parameter values, my `Likelihood`s will often return `java.lang.Double.NEGATIVE_INFINITY` since $-\infty$ is the log of zero. This will have the property that transitions outside the support of the posterior distribution will be rejected (provided that the chain starts inside the support: failing to do so is a fairly common error). This is one way of preserving the positivity of variance parameters and so forth: it is more elegant to prevent parameter values with zero posterior probabilities from being proposed, but this technique works.

Before the new likelihood function can be used, it must be compiled (e.g. using `javac MyLikelihood.java`) to create a file called `MyLikelihood.class`. This class must be visible in your classpath.

2.9 Writing your own ArgumentMakers

It is likely that you will have the occasion to write your own `ArgumentMaker` functions before needing to write any new `Likelihoods`. The definition of an `ArgumentMaker` is a class that contains a method called `getArgument` that takes a two-dimensional array (not necessarily rectangular) of `double`s as input and returns a one-dimensional array of `double`s. The input array will be the values of all the parameters in the bond, and the output will be the values to send as one of the arguments to the Likelihood. The `ArgumentMaker`'s constructor can accept several inputs if constants or constant vectors are useful in calculating the argument. For example, expanders in `GroupArgument` are inputs to the constructor of that class. Covariates in `LinearModelArgument`, in the form of a data frame and indicators of which columns correspond to numerical or categorical predictors, are inputs to that class. Another common technique is to send an integer to the constructor that indicates which parameter plays a certain role. The simplest example is in the `IdentityArgument` class: its constructor takes a single integer as an argument, and this integer points to the parameter whose value will be returned unchanged.

3 Enhancing mixing

Inevitably, frequent users of YADAS will run into situations where the default algorithms based on componentwise random walk Metropolis–Hastings steps are inadequate for generating MCMC algorithms that mix appropriately well. While the theorems guarantee that the limiting distribution of the chain is the desired posterior distribution, the consecutive samples may be too highly correlated for efficient inference. The most common reason for this is that two or more parameters are highly correlated, so that those parameters cannot move freely individually.

Other MCMC practitioners may try other techniques for improving mixing. For instance, reparameterization so that parameters are more nearly uncorrelated can work, and Gibbs sampling experts will often try Gibbs updates of vectors of parameters (in which one samples a vector of parameters from their conditional distribution given the values of the other parameters; see C. Liu, 2003, “Alternating Subspace-Spanning Resampling to Accelerate Markov Chain Monte Carlo Simulation”, to appear in JASA, for a recent approach). Parameter expansion data augmentation (JS Liu and Y Wu (1999) ”Parameter expansion for data augmentation”, JASA 94, 1264-1274) is a further exciting approach. These techniques are possible within YADAS in the sense that you can write all the necessary code and add it to your analysis. The approach that we follow is more in the spirit of YADAS: making additional Metropolis-Hastings moves in which proposals attempt to change multiple parameters simultaneously.

3.1 MCMCUpdate

`MCMCUpdate` is a YADAS interface consisting of four methods, the most important of which is `update()`. (The others, `accepted()`, `updateoutput()`, and `finish()`, are related to acceptance probability output and it is not necessary to define them to be anything other than empty methods.) Calling the `update()` method of an object implementing the `MCMCUpdate` interface will attempt to modify one or more of the parameters in the algorithm. You have already seen examples of `MCMCUpdate`s, since each parameter is itself an update. What this means is that if `theta` is an `MCMCParameter` with (say) K components, calling the method `theta.update()` loops over the components of `theta`, attempting to change the kth component of `theta` from its current value θ_k to $\theta_k + s_k Z_k$, where Z_k is a standard Gaussian random variable. Each componentwise move is accepted if the ratio of the posterior distribution evaluated at the new value, to the posterior distribution evaluated at the old value, is greater than a uniform$(0, 1)$ random variable. A YADAS analysis includes an array of objects implementing the `MCMCUpdate` interface. The first attempt for an analysis generally constructs this array by listing the `MCMCParameter`s in the analysis.

In the remainder of this section, we discuss another class implementing the `MCMCUpdate` interface, the `MultipleParameterUpdate` class. (Other examples of update classes are presented in the next section: `ReversibleJumpUpdate` in §4.2, and `FiniteUpdate` in §4.1.) Before concluding it is necessary to use nonstandard updates, it makes sense to tune the Metropolis step sizes as well as possible. YADAS applications generally send acceptance rate information to standard output. Returning to Example 1, the one way ANOVA example contains four `MCMCUpdate`s, `mu`, `theta`, `sigma`, and `delta`. For our example data file, `mu` has three components and the others are all one-dimensional. After I ran this application for 11000 iterations, the following was sent to standard output:

```
Update 0: 0:4266 1:4279 2:4322
Update 1: 0:5571
Update 2: 0:3809
Update 3: 0:4104
```

These are the numbers of accepted Metropolis moves out of 11000 attempts: ’Update 0’ refers to `mu`, which has three components, and hence three distinct update steps. Metropolis changes to the values of `mu` were accepted respectively 4266, 4279, and 4322 out of 11000 attempts. Acceptance rates for `theta` were higher, while they were lower for `sigma`. These acceptance rates can be used to tune the step sizes. We shoot for acceptance rates of 40% and are normally quite happy with anything within 15% of that in either direction. A. Gelman, G. O. Roberts, and W. R. Gilks ("Efficient Metropolis jumping rules", in Bayesian Statistics 5, 1995) did theoretical work in a univariate normal problem showing that acceptance rates of 15-50% were near optimal. Whether or not this theory can be extended to problems with more dimensions, we appear to get good results through aiming for these acceptance rates.

However, sometimes acceptance rates remain low even when the step size is small in comparison to the posterior standard deviation of the parameter. Time series plots of parameters can be helpful in diagnosing this problem, and normally the explanation is that the parameter is highly correlated with some other parameter. This is where `MultipleParameterUpdate`s come in to YADAS analyses of difficult problems.

3.2 MultipleParameterUpdates

The `MultipleParameterUpdate` class is itself simple to use: the only arguments to its constructor are an array of `MCMCParameter`s that the update will attempt to move simultaneously, and an object implementing the `Perturber` interface that indicates the function to apply to the parameters to generate the proposed move. As a simple example, suppose that the posterior distribution of interest is a bivariate normal distribution with high correlation: θ_1 and θ_2 have mean zero, $\mathrm{Var}(\theta_i) = \sigma_i^2$, and $\mathrm{Corr}(\theta_1, \theta_2) = \rho$. Alternating between Metropolis (or Gibbs) moves to θ_1 and to θ_2 may lead to a slow mixing Markov chain. However, large changes to both parameters can be obtained by adding a Metropolis move in which the new proposed value (θ_1', θ_2') satisfies $\theta_1' = \theta_1 + sZ$ and $\theta_2' = \theta_2 + s(\sigma_2/\sigma_1)Z$, where Z is a standard normal random variable independent of everything and where $s > 0$ is an appropriate (and, in YADAS, easily tunable) step size. In other words, `MultipleParameterUpdate`s deal with correlation by adding a Metropolis or Metropolis-Hastings step in which we attempt to move the parameters in a direction of high, if not necessarily exactly maximal, variability. These directions can most often be intuited by considering the form of the posterior distribution: in fact, in most cases in our experience, a single term in the posterior distribution is the culprit.

The difficult part of adding code for a `MultipleParameterUpdate` to an analysis is writing the code for the perturber or finding an appropriate existing perturber (the library of perturbers is not organized, and I find myself writing perturbers with the same functionality multiple times). We will illustrate the issues with some examples.

3.3 Example 4: NewAddCommonPerturber

In this example, we consider another one-way ANOVA problem in which the data variance is considerably larger than the variance of the random effects.

For illustrative purposes, we use the "wrong" parameterization. It is well known (see, e.g. GO Roberts and SK Sahu, JRSSB, 1997, 59:291-317) that the model $Y_{ij} \sim N(\alpha + \theta_i, \sigma^2), \theta_i \sim N(0, \sigma_\theta^2)$ mixes well in the Gibbs sampler under these conditions on the variances, whereas the model $Y_{ij} \sim N(\mu_i, \sigma^2), \mu_i \sim N(\theta, \delta^2)$ mixes poorly. We work with the poorly mixing model in order to demonstrate how the `MultipleParameterUpdate` fixes the mixing difficulties. The first half of the MCMC iterations for this example are performed with the special update, and the last half include it. A time series plot of the iterations of μ_i or θ will illustrate the difference. The mixing problem arises because the μ_i's are highly correlated with each other and with θ. In fact, one can add the same constant to each of these parameters without changing the value of the $\mu_i \sim N(\theta, \delta^2)$ bond, and since δ is small, this is the dominant bond in some sense. We propose a move that leaves the value of this bond fixed and allows the other bonds (since the prior for θ is flat, the only relevant bond is the data bond) to decide whether the move should be accepted. This move adds the same random Gaussian random variable to θ and to all of the μ_i. Hence, it is called a `NewAddCommonPerturber`.

To define a `NewAddCommonPerturber`, one supplies two arguments to its constructor: a two-dimensional array of integers that should have the same shape as the array of parameters being updated, and a one-dimensional array of real step sizes. The integers should range from zero up to one less than the length of the step size array. A `NewAddCommonPerturber` defines as many update steps as there are values in the step size array. The first update step identifies all zeroes in the array of integers, generates a random Gaussian variable with standard deviation equal to the first step size, and adds this random variable to the parameter entries in the same position as the zeroes in the integer array. If the step size array has more than one element, the second step does the same thing with all the ones in the integer array and the second step size. For example, suppose that the parameters being updated are μ, θ, and γ, where μ has length eight, θ has length four, and γ has length two. If the integer array is

$$\{\{0,0,0,0,1,1,1,1\},\{0,0,1,1\},\{0,1\}\},$$

and the step size array is $\{1.5, 0.5\}$, the first proposed update will take $Z_1 \sim N(0,1)$, and let $\mu_i' = \mu_i + 1.5Z_1(i = 0,1,2,3), \theta_i' = \theta_i + 1.5Z_1(i = 0,1)$, and $\gamma_0' = \gamma_0 + 1.5Z_1$. After it is decided whether or not to accept this move, the second proposed update will be to take $Z_2 \sim N(0,1)$ and let $\mu_i' = \mu_i + 0.5Z_2(i = 4,5,6,7), \theta_i' = \theta_i + 0.5Z_2(i = 2,3)$, and $\gamma_1' = \gamma_1 + 0.5Z_2$.

`NewAddCommonPerturber`s are likely to be useful in a wide variety of hierarchical models, and if the hierarchical model has more levels, more than one special update may be necessary. `NewAddCommonPerturber` is not as useful as it might be; for example, it doesn't allow single parameter values to be changed in more than one step.[1] The older class `AddCommonPerturber` allows this but is harder to use.

3.4 Example 5: NewOneUpOneDownPerturber

In this example, the model suffers from poor identifiability in the sense that the likelihood is unaffected by a parameter transform in which we add a constant to one parameter and subtract the same constant from other parameters. To be precise, we are working with a one-way ANOVA problem again, this time using the non-centered parameterization: the data y_{ij} are distributed as $N(\mu + \alpha_i, \sigma^2)$, where μ has a flat prior, $\alpha_i \sim N(a_\alpha, b_\alpha^2)$, where a_α, b_α, and σ are all known. This is a stripped down example to illustrate the issues clearly.

When we run this example with the MCMC algorithm consisting only of componentwise random walk Metropolis updates of μ and α, the algorithm performs poorly because each of the α_i's is highly negatively correlated with μ. We solve this problem with a `NewOneUpOneDownPerturber`, which proposes a change to the parameters as follows: let $Z \sim N(0,1)$, and let $\mu' = \mu + sZ$, $\alpha_i' = \alpha_i - sZ$ for all i. The constructor to `NewOneUpOneDownPerturber` accepts two arguments: first, a two-dimensional array of integers, and second, an array of real-valued step sizes. The interpretation of the two-dimensional array is much the same as in `NewAddCommonPerturber`: the zeroth attempt at updating the parameters adds sZ (where s is the zeroth step size) to all the coordinates in the zeroth parameter that are identified with a zero in the zeroth column in the array of integers and subtracts sZ from all the coordinates in the first parameter that are identified with zeroes in the first column of the array of integers. The next update looks for ones in the two-dimensional array, etc. In the `NoncenteredANOVA` example, all of the parameters (μ and all α_i) are being updated together, so all of the entries in the two-dimensional integer array are zero.

3.5 Writing your own Perturbers

Partly because I have been unable to get my `Perturber` libraries organized, if you want to use a `MultipleParameterUpdate` in your own application, there is an excellent chance that you will have to write your own class that implements the `Perturber` interface. This interface consists of three methods.

[1]Alert readers will observe that it is in fact possible to do this if the array of parameters contains the same parameter multiple times.

- `perturb()` takes as inputs a two-dimensional array of real numbers and an integer, and returns nothing (although it does change the values inside the array of reals). The array represents current values of some of the parameters, and the purpose of the `perturb()` method is to change them to proposed values. The integer pertains to the case where the update class needs to generate several successive update steps; the integer communicates how far along on this sequence the algorithm has gotten (starting with zero, of course). The only restrictions on what changes the `perturb()` method makes to the array's values are that it should be possible to define an appropriate `jacobian()` method as described below.

- `numTurns()` takes no arguments and returns an integer, the number of separate update steps that this perturber is responsible for generating. It is common for a `MultipleParameterUpdate` to consist of only one such step, in which case `numTurns()` returns 1. Otherwise, a `Perturber` will often require an array of step sizes as inputs, and `numTurns()` will return the length of this array. The `whoseTurn` variable that is an argument to `perturb()` ranges from zero to `numTurns() - 1`.

- `jacobian()` takes no arguments and returns the Hastings adjustment $\frac{T(\theta',\theta)}{T(\theta,\theta')}$ in the notation of §1.1.[2] In the case of additive Gaussian proposals to the parameters as in `NewAddCommonPerturber` and `OneUpOneDownPerturber`, this ratio is equal to 1.0. Another case I use is `ScalePerturber`, which is like the additive adjustments but on the log scale: the proposal consists of several parameters being multiplied by $r = \exp(sZ)$, where $s > 0$ is a step size and Z is standard Gaussian. In this case, `jacobian()` returns r raised to the power of the number of parameters being changed. The number that `jacobian()` returns will often be computed inside `perturb()` and stored inside an instance variable.

I make a big deal about how YADAS saves users from having to evaluate full conditional distributions, so it is disappointing that the Hastings ratios require user evaluation. I have much experience in getting the ratio wrong, and it is most often quite obvious from the MCMC output that the algorithm is not stationary.

3.6 Writing your own Updates

YADAS's open source distribution and unrestrictive architecture make it possible for you to augment it with any update method you choose. For example, it is possible for special applications to include Gibbs updates of some parameters in the event that complete conditional distributions are available and Metropolis algorithms are for whatever reason not working. The only Gibbs update we have seen fit to implement is the case of a parameter that takes on finitely many values; see §4.1 below.

4 Advanced topics and examples

In this section we discuss several "advanced topics" that are used only in special-purpose YADAS applications (though some have potential to attain more widespread usage). First, we present how to update parameters that take on finitely many values. Next, we discuss reversible jump MCMC and apply it to an example where we test a hypothesis that two binomial proportions are equal. Then we present an example related to auto racing that has an unusual likelihood function, and finally we study the reliability of a complex system based on information at the component, subsystem, and full system level.

[2]It is probably also another example of a poorly named method or class in YADAS.

4.1 FiniteUpdate, and Example 6: BetaBinomialExample

In this section we learn about how to update parameters that take on finite numbers of values. Clearly, it does not work to add a Gaussian step to such a parameter. Instead, we address this problem by sampling from the full conditional distribution of the discrete parameter. (This is as yet the only example where we have allowed YADAS to be polluted with sampling from full conditional distributions.) YADAS's infrastructure for computing ratios of posterior distributions serves it well in this situation. Suppose $f(x, \theta)$ is the unnnormalized posterior distribution evaluated at a discrete-valued parameter x and the remaining parameters θ. If the current value of x is i and we are contemplating changing the value of this parameter to j, YADAS is set up to compute the ratio $r_j = f(j, \theta)/f(i, \theta)$. If we compute this ratio for all values of j, a sample from the full conditional distribution of x chooses $x = j$ with probability proportional to r_j. $f(i, \theta)$ is wastefully computed several times, but one should worry about this only in rare circumstances.

It is quite easy to use the `FiniteUpdate` class: one simply inserts a `FiniteUpdate` into the update array. The constructor of `FiniteUpdate` requires only the name of the parameter and an array of integers with the same number of elements as the parameter. The integers list how many possible values each element of the parameter can take on. (We assume that the possible values of the ith element of the parameter are $\{0, 1, \ldots, n_i - 1\}$, in which case n_i should be placed inside the integer array.)

We illustrate the use of `FiniteUpdate` with a simple example in which we sample from the beta-binomial distribution. This example was inspired by G. Casella and EI George, (1992) "Explaining the Gibbs Sampler", *The American Statistician* **46**:3:167-174. This is not a statistical problem in that there is no data: the situation is that a probability y is sampled from a Beta(a, b) distribution, and conditionally on y, x is sampled from a binomial distribution with sample size n and probability y. We estimate the joint distribution of (y, x) using MCMC: we alternate between updating y and x, updating y when it is its turn using random walk Metropolis, and updating x by sampling from its full conditional. As noted in the code, the code has a few annoying features: the input file must contain a real variable n to use as an argument to the `Binomial Likelihood`, and it also must contain an integer variable $ni = n + 1$ to tell `FiniteUpdate` the number of possible values of x. A sanity check is that the marginal distribution of y turns out to be the beta distribution with the supplied parameters.

Note that the restriction that the possible values of the parameter are zero up to some maximum is in fact not a restriction at all, because such parameters are ideally suited for use as subscripting variables in `ArgumentMaker`s.

In some cases, the number of possible values of a finite-valued parameter is large, so it may be inefficient to entertain all possible values when updating the parameter. In this case, `IntegerMCMCParameter` can be used to propose a Gaussian-like step that proposes a discrete move. The constructor of an `IntegerMCMCParameter` looks exactly like that for an ordinary `MCMCParameter`, so the step size is the standard deviation of the Gaussian that is sampled before being converted to an integer. (The proposal is $\theta' = \theta + \text{sgn}(Z)\{1 + \text{floor}(|sZ|)\}$, where Z is standard normal.) If you use this option, there must be a likelihood function that prevents θ from taking on disallowed values, using negative infinities.

4.2 ReversibleJumpUpdate, MixtureBond, and Example 7: BinomialHypothesisTest

One of the most challenging sitations when implementing an MCMC is if a parameter has a mixture distribution and is most naturally thought of as living in one of two different spaces. A simple example arises in variable subset selection in regression: a slope parameter may be given a prior distribution that is a mixture of a normal distribution and a point mass at zero. In some cases it will be possible to update such a parameter using its full conditional distribution, but in other cases this distribution is not tractable, so that we want to use some method more like Metropolis-Hastings. Reversible jump MCMC, as presented by PJ Green

("Reversible Jump MCMC Computation and Bayesian Model Determination", 1995, Biometrika 82 711-732) demonstrates how to do this. The algorithm attempts to jump between spaces, and the Metropolis-Hastings acceptance probability is a function of the methods of proposing new parameter values so that the stationary distribution of the algorithm is as desired.

The `ReversibleJumpUpdate` can be thought of as a transition matrix between models, and an array of functions specifying how to propose a new value of the parameters in a new model, as a function of which model was the old model. For this to work, it is necessary to adjust acceptance probabilities in such a way that the algorithm's limiting distribution is the desired posterior distribution. The arguments to the constructor are

- an array of parameters whose values might be changed by the reversible jump update;
- an integer that determines how many models we are mixing over (denote this integer by M);
- an integer that indicates which of these models is the initial state for the MCMC (this should be consistent with the initial values of the parameters, and its possible values are $0, 1, \ldots, M-1$);
- a one-dimensional array of transition probabilities between the models. This array will be interpreted as a square $M \times M$ matrix whose (i, j) coordinate is the probability that if the current model is model i, the next model proposed will be model j;
- an array of `JumpPerturber`s. A `JumpPerturber` is much like a `Perturber`. Both are interfaces, and `JumpPerturber` extends `Perturber` by adding a `density()` method. This method computes the probability density of the proposed new value of the parameters.
- a String indicating a directory where output specific to the reversible jump update should go. This output includes acceptance probabilities for the purpose of Rao-Blackwellization, but has not been used enough to be standard.

To be precise, suppose the current model is model i and the current value of the parameter is θ. To generate a new value of the parameter, we first choose a new model according to the probabilities in the transition matrix: suppose model j is chosen; the probability that this happens is (say) A_{ij}. The proposed new value θ' of the parameter is chosen from density $T_{ij}(\theta, \theta')$. Denote the unnormalized posterior by f. The acceptance probability for the reversible jump move is

$$\frac{f(\theta')}{f(\theta)} \frac{A_{ji} T_{ji}(\theta', \theta)}{A_{ij} T_{ij}(\theta, \theta')}.$$

This is "just" the Metropolis-Hastings rule, but specialized to include the possibility of jumping across models. (The `JumpPerturber`s evaluate the densities $T_{ij}(\theta, \theta')$.)

In the following example, we perform a Bayesian hypothesis test for equality of two probabilities p_1 and p_2. We have binomial samples $x_i \sim \text{Binomial}(n_i, p_i)$ for $i = 1, 2$. We make it into a hypothesis testing problem by putting a mixture prior on (p_1, p_2): with probability λ, $p_1 = p_2$ and their common value has a beta prior distribution, while with probability $1 - \lambda$, the p_i's are exchangeable, each with a (possibly different) beta prior distribution. To follow this example, it is necessary to understand the `MixtureBond` class. A `MixtureBond` is essentially an array of `BasicMCMCBond`s, one bond for each model that we are mixing over. While a `BasicMCMCBond` requires an array of parameters, an array of `ArgumentMaker`s and a `Likelihood`, a `MixtureBond` requires two-dimensional arrays of parameters and `ArgumentMaker`s and an array of `Likelihood`s. In addition, a `MixtureBond` requires a single `ArgumentMaker` that computes the mixing probabilities for each of the models. This is implemented in a slightly strange way: when this `ArgumentMaker` is applied to the parameters for the mth model, it should generate an array whose mth coordinate is the probability of the mth model. The simplest way to do this is with a `ConstantArgument`, but we wanted to enable the case where the mixing probabilities are themselves being estimated. We emphasize that a mixture bond is used for computing weighted averages of bonds. Suppose $x_1, x_2, \ldots x_n$ may come from several densities f_m, and each of these distributions has probability π_m. A single `MixtureBond` can be used to compute

$$\sum_m \pi_m \prod_i f_m(x_i),$$

but not to compute

$$\prod_i \sum_m \pi_m f_m(x_i).$$

The latter can be computed using n `MixtureBond`s. In other words, a single `MixtureBond` mixes the distributions in such a way that all of the data points come from the same bond. You may want the other interpretation, and I intend to produce such a class as soon as possible. In short, one should expect some transience in the code for mixture distributions in YADAS.

To get (finally) to the example, the single unknown parameter p contains both the probabilities p_1 and p_2. The `MixtureBond` contains the mixture distribution and is used in a somewhat nonstandard way here. If β_{ab} denotes the beta density with parameters a and b, the mixture bond is intended to compute the quantity

$$(1-\lambda)\beta_{ab}(p_1)\beta_{ab}(p_2)I_{\{p_1 \neq p_2\}} + \lambda\beta_{ab}(p_1)I_{\{p_1 = p_2\}}.$$

The `AreTheyEqualArgument` does the work here: if $p_1 = p_2$, this argument generates the array $\{0, \lambda\}$, while if $p_1 \neq p_2$, the output is the array $\{1-\lambda, 0\}$, since the first model is the $p_1 \neq p_2$ model. This is a nonintuitive trick but it will presumably be common in mixture models that require `ReversibleJumpUpdate`s (depending on the parameter, one of the models may have zero probability).

Finally, the parameter p is updated using reversible jump. The matrix ($a_{00} = 0.75, a_{01} = 0.25, a_{10} = 0.25, a_{11} = 0.75$) indicates that if $p_1 \neq p_2$, the probability is 0.25 that the proposed new value of p will feature $p_1 = p_2$. Also, if $p_1 = p_2$, the next proposed value will set them unequal with probability 0.25. The four `JumpPerturber`s operate as follows.

- If the p's are unequal and the proposed new values are also to be unequal, we add a different Gaussian random walk step to each on the logit scale.
- If the p's are unequal and the proposal is to equalize them, the proposal is deterministic and equal to a weighted average of the two p's. The weights are set in the input file.
- If the p's are equal and are to be made unequal in the proposal, we again add different independent Gaussians to the p's on the logit scale.
- If the p's are equal and they should be kept equal in the proposal, we update their common value by adding a Gaussian on the logit scale.

The standard output from a `ReversibleJumpUpdate` is a natural extension of the usual acceptance probability output; this time it shows the number of acceptances and trials from each model to each model. I obtained the following output:

```
Update 0: 0 -> 0: 1073 / 1551; 976 / 1551;
0 -> 1: 443 / 496;
1 -> 0: 442 / 2200;
1 -> 1: 4761 / 6753;
```

This means that when the algorithm moved from model 0 to model 0, the first p was changed successfully 1073 out of 1551 times, while the second p was changed 976 out of 1551 times. The acceptance probability was quite high when we proposed a move from $p_1 \neq p_2$ to $p_1 = p_2$, but low in the opposite direction.

Theoretical results about ideal acceptance rates, values of the transition matrix and choice of the `JumpPerturber`s would be of great interest and probably not easy to achieve.

4.3 Example 8: AttritionLikelihood

I could hardly create a web site about YADAS without including my favorite example, especially since it was in fact the motivating example for its development. For those who are uninterested in the subject matter, this example still highlights some of the power and versatility of YADAS and especially the `BasicMCMCBond` construct.

Suppose that one wishes to analyze data that represent several permutations of subsets of individuals. Our example (see T. Graves, C. S. Reese, and M. Fitzgerald, "Hierarchical Models for Permutations: Analysis of Auto Racing Results", to appear in *Journal of the American Statistical Association*; until then find it at http://madison.byu.edu/racing/racing.html) is the finishing orders of drivers in a set of stock car races. We wished to address several questions using these data, including how to construct overall measures of driver ability, whether some race tracks had results that were more predictable than others, and to what extent drivers' abilities differed from track to track. To this end let θ_{ij} denote the ability of driver i in race j. The probability distribution of the finishing order in race j depends on the parameters θ_{ij} in the following way. First, the last place finisher in the race is chosen from the participants with probability proportional to $\lambda_{ij} = \exp(-\theta_{ij})$. Next, the second-to-last place finisher is chosen from the remaining drivers with probability proportional to the λ_{ij}, and the process is continued until two drivers i_1 and i_2 remain, and driver i_1 finishes second with probability $\lambda_{i_1 j}/(\lambda_{i_1 j} + \lambda_{i_2 j})$. (This formulation is equivalent to drawing independent exponential random variables for each driver with mean $\exp(\theta_{ij})$, and setting the finishing order to be decreasing in the exponential variables.) Models of interest for the θ's include

- the case where a driver's ability $\theta_{ij} = \theta_i$ is the same in every race,
- driver abilities are changing over time,
- some tracks are more predictable than others in the sense that they intensify the importance of driver abilities, i. e. $\theta_{ij} = \theta_i \phi_{T(j)}$, where $T(j)$ denotes the track of race j,
- track-driver interactions are important, i. e. $\theta_{ij} = \theta_i + \Omega_{iT(j)}$.

The key construct for analyzing any of these models is the `AttritionLikelihood`. Its task is to accept an array of driver ability parameters θ_{ij} and an array of track predictability parameters ϕ_j (these last are generally equal to one) and return the log of the value of the likelihood

$$\prod_{j=1}^{J} \prod_{i=1}^{I_j} \lambda_{\pi_j(i),j} \left(\sum_{k=1}^{i} \lambda_{\pi_j(k),j} \right)^{-1} .$$

Here J is the number of races, I_j is the number of drivers in race j, π_j is defined so that $\pi_j(k) = i$ means that driver i finished in kth place in race j, and $\lambda_{ij} = \exp(-\theta_{ij}\phi_j)$. To do this, its constructor requires two arrays of integers: an array of race identifiers (ranging from zero to one less than the number of races) and an array of finish position identifiers (ranging from one to I_j). The special thing about this `Likelihood` is that it breaks away from the usual structure in which each "row" in the two-dimensional array is handled separately (e.g. $y_i \sim N(\mu_i, \sigma_i^2)$). To calculate the contribution to the likelihood of the tenth place finisher in the first race, one needs to know the identities of the first nine finishers as well.

The code for this application is in `AttritionAnalysis.java`. This application analyzes the model in which driver abilities do not depend on the race, and the track predictability parameters are identically equal to one. The model specification is completed by assuming that driver abilities $\theta_i \sim N(0, b_\theta^2)$[3],

[3]Actually the code contains a parameter a_θ to symbolize the mean of the θ_is, but this parameter is fixed at zero.

where b_θ has a Gamma (exponential, in fact) distribution. Convergence is improved through the use of a `MultipleParameterUpdate` with a `ScalePerturber` that attempts to rescale simultaneously the θ_i and b_θ. Note that one of the richer models for θ_{ij} can be obtained by using different `ArgumentMaker`s: the desire to be able to fit several models with minimal changes to the application code motivated the design of the `ArgumentMaker` class.

The input files contain data from the 2002 NASCAR Winston Cup season, which we obtained from http://www.nascar.com. The file `attw02.dat` contains four integer columns: the race ID and finishing position ID to be used as inputs to the `AttritionLikelihood` constructor, and the identifiers of the driver and track. One good feature of the decision to require users to begin their input files with the number of lines of data is that after the last line, users can add comments: here we list the correspondence between driver IDs and driver names, and between track IDs and track names. The interpretation of the first line of data, "0|1|86|6", is that in the first (zeroth) race, the first place finisher was driver 86, Ward Burton, and the race was held at track 6, Daytona. (This race was the 2002 Daytona 500.) The file `mssw02.dat` contains Metropolis step sizes for the driver ability parameters θ_i; we have found that it usually works well to use 0.2 multiplied by the square root of the number of races in which the driver participated.

4.4 Example 9: GenericSPSystem

We move from the most frivolous example yet implemented in YADAS to one that is a key example of the analyses we work on at Los Alamos. This example involves estimating the reliability of a system, where the system is composed of subsystems and components, test data is potentially available on components, subsystems, or the entire system, and expert opinion is also available at any of those levels. This methodology is discussed in VE Johnson, TL Graves, MS Hamada, and CS Reese, "A Hierarchical Model for Estimating the Reliability of Complex Systems," in Proceedings of Seventh Valencia Conference on Bayesian Statistics, though here we have changed the system structure and the data.

The problem is depicted by the graph in the Figure on this page. The interpretation of the graph is as follows: the bottom row of nodes (nodes three through seven) are leaf nodes and they represent components that can succeed or fail independently. The statistical model assumes that the success probabilities (reliabilities) for these components are p_3, p_4, p_5, p_6, and p_7. Moving up the tree means that components are integrated into subsystems either in parallel or in series. The subsystem given at node 2 represents components five through seven being integrated in series. In other words, the probability that a test of the subsystem at node 2 succeeds is $p_2 = p_5 p_6 p_7$. The subsystem at node one involves nodes 3 and 4 combining in parallel, and this subsystem succeeds with probability $p_1 = 1 - (1 - p_3)(1 - p_4)$. Finally, the system at node 0 combines the two subsystems in series, so it succeeds with probability $p_0 = p_1 p_2 = \{1 - (1 - p_3)(1 - p_4)\} p_5 p_6 p_7$. We may have test data at any node, and the probability that a test of node i succeeds is p_i. We may also have expert opinion about the reliability of any node, and we interpret this as if it were data: i.e. we translate the statement that an expert believes node i to have a beta distribution with parameters $\nu_{g(i)} \tilde{p}_{h(i)}$ and $\nu_{g(i)}(1 - \tilde{p}_{h(i)})$ into $\tilde{p}_{h(i)}$ successes in $\nu_{g(i)}$ binomial trials with success probability p_i. If desired, the $\tilde{p}$'s and ν's can be given prior distributions, and this has some advantages; see the Johnson et al paper. It is important to node that the p_i for the leaf nodes (components) constitute parameters of the model, but the other p_i are simply functions of these parameters. In particular, when we incorporate expert opinion on a non-leaf node, we do so by including the appropriate function of the leaf reliabilities in the place of the binomial success probability.

The `GenericSPSystem` class[4] facilitates analysis of most problems of this form in which components and subsystems are combined in parallel to form a system. A restriction is that the system must be tree-structured, i.e. one component cannot contribute to more than one subsystem. The first unusual code in this example is the definition of `system`, an instance of the class `ReliableSystem`. This line simply

[4]The class is named after a predecessor called GenericSeriesSystem, and the new class also handles components integrated in parallel.

constructs an empty system with the appropriate number of nodes. The next line reads the graph structure from the file `components.dat`, a file that contains the parents of the nodes in the graph in an integer column called 'parents': here the parent of a node is the node that represents the larger subsystem that it is a part of. The 'gate' column indicates whether the subsystem is obtained by a parallel (gate ≤ 0) or series (gate > 0) integrator. The first bond in the system incorporates the test data. The interesting `ArgumentMaker` is obtained by a method `fillProbs` of the `ReliableSystem` class. The function of this method is to take the values of p_3, p_4, p_5, p_6, and p_7 as input and return an array consisting of $(\{1-(1-p_3)(1-p_4)\}p_5p_6p_7, \{1-(1-p_3)(1-p_4)\}, p_5p_6p_7, p_3, p_4, p_5, p_6, p_7)$. We also use this `ArgumentMaker` in the second bond that represents the expert opinion: the vector of all the p_i's are treated as coming from Beta distributions with parameters $\nu\tilde{p}+1$ and $\nu(1-\tilde{p})+1$, which is equivalent to incorporating them as binomial data. The p parameter was defined as a `LogitMCMCParameter` so that its components are updated on a logit scale, and this choice leads to acceptable mixing for all the unknown parameters.

5 License agreement